Karlsruhe Institute of Technology

Institute for Data Processing and Electronics

Practical Microsecond Real-Time Reinforcement Learning

Luca Scomparin, Michele Caselle, Andrea Santamaria Garcia, Chenran Xu, Timo Dritschler | 5-7 February 2024

Table of contents

Karlsruhe Institute of Technology

1. Introduction & Motivation
   - Real-Time and its constraints
   - Why is Real-Time AI hard?
2. Alternative computing platforms
3. Challenges of reusing existing implementations
4. Real-Time RL: experience accumulator and the KINGFISHER platform
5. Is μs Real-Time RL what you need?
6. Conclusion

Motivation

Great data rate $\rightarrow$ lot of training data

Motivation

Great data rate $\rightarrow$ lot of training data

Possibility of **training online**

Motivation

Karlsruhe Institute of Technology

Great data rate $\rightarrow$ lot of training data

Possibility of **training online**

Timing constrains become relevant!

What is Real-Time?

Shin and Ramanathan (1994) identify major components:

> *Correctness of a computation depends not only on the logical correctness but also on the time at which the results are produced.*

1. "time" is the most precious resource;
2. reliability is crucial;
3. **environment of operation** is an active component.

Predictability is fundamental!

Three possible levels/categories:

1. *hard*, catastrophic consequences;
2. *firm*, results produced late not useful;
3. *soft*, later means decreasing usefulness.

Depending on environment, RL can be either one of these!

Latency and throughput constraints

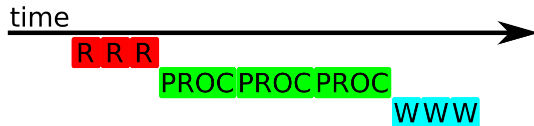
Karlsruhe Institute of Technology

Latency

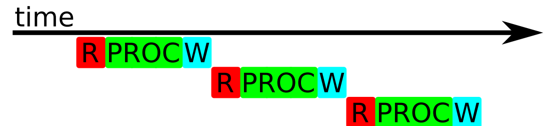
Time needed to produce output after input is received

Throughput

Maximum rate at which data can be processed



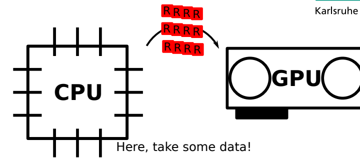
Bad latency, batched

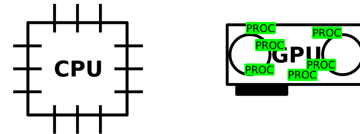


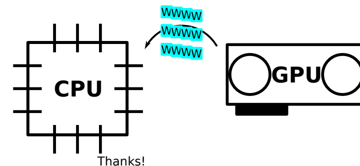
Good latency, streaming

Issues of Real-Time AI

- Current ML frameworks have mainly throughput in mind $\rightarrow$ no/little real-time optimization;
- use of batched execution on GPU $\rightarrow$ not optimal for latency;
- conventional computing hardware not meant for low-latency real-time;
- it still works great for latency in the millisecond range!





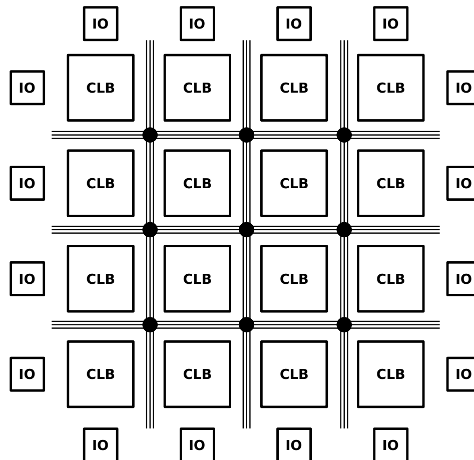


FPGAs

FPGA → Field Programmable Gate Arrays

- lattice of logic blocks;
- "flow" of computation is user defined;
- extremely low overhead;
- not as high-performance as CPUs or GPUs.

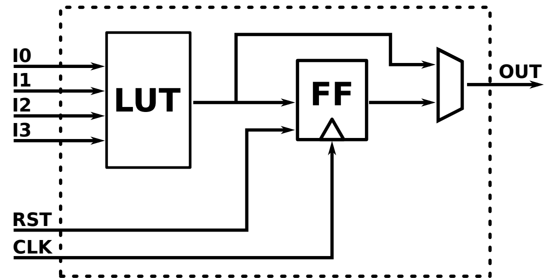
Easier to enforce latency and throughput constraints!
Not as trivial to program



FPGAs

FPGA → Field Programmable Gate Arrays

- lattice of logic blocks;
- "flow" of computation is user defined;
- extremely low overhead;
- not as high-performance as CPUs or GPUs.



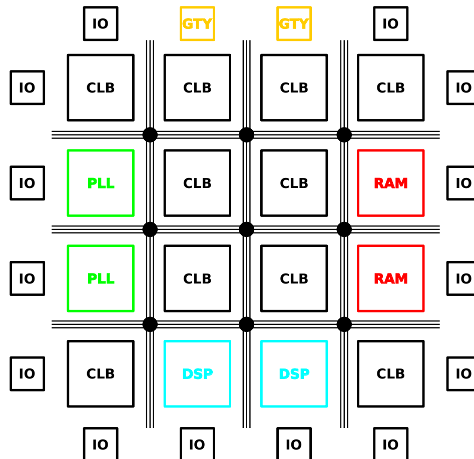
Easier to enforce latency and throughput constraints!
Not as trivial to program

FPGAs

FPGA → Field Programmable Gate Arrays

- lattice of logic blocks;
- "flow" of computation is user defined;
- extremely low overhead;
- not as high-performance as CPUs or GPUs.

Easier to enforce latency and throughput constraints!
Not as trivial to program

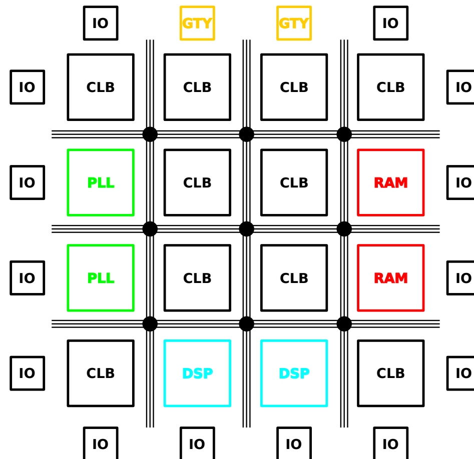


FPGAs vs GPUs

Karlsruhe Institute of Technology

From Rothmann and Porrmann (2022):

- GPUs good for training of DNNs;
- RL algorithms use many GPU kernels with little computation;
- increased launch over-head.



Heterogeneous platforms

Karlsruhe Institute of Technology

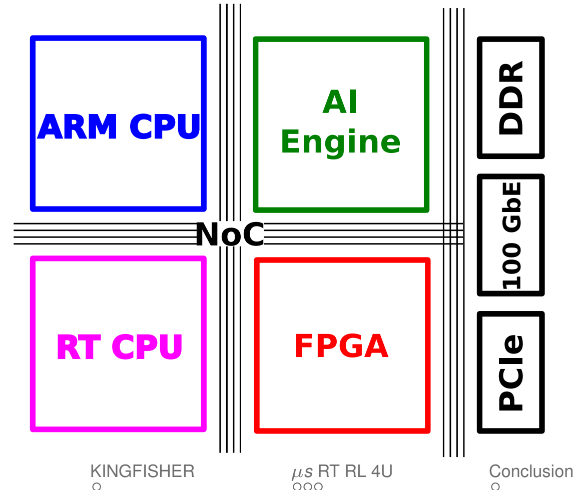
Different computing platforms $\rightarrow$ different benefits

Heterogeneous combine CPUs, FPGAs and "GPUs"

An example, AMD Versal:

- combines FPGAs and ARM CPUs;
- AI Engine array for heavy multiplication workloads;
- Network-on-Chip interconnect;
- high-speed interfaces.

These computation unit work in synergy and share memory!



Why can’t we reuse code?

Karlsruhe Institute of Technology

Instructions of different platforms are different!

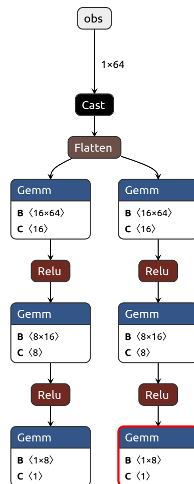
There are several NN implementations. BUT:

- mostly throughput optimized;
- targeting different platforms;
- real-time not in mind.

Most NN deployment to FPGA quantize $\rightarrow$ effect on RL algorithms?

Would be cool to directly translate agents into real-time, but we are not there yet.

An ONNX middle-layer for low-latency could be the solution!



Why can't we reuse code?

Instructions of different platforms are different!

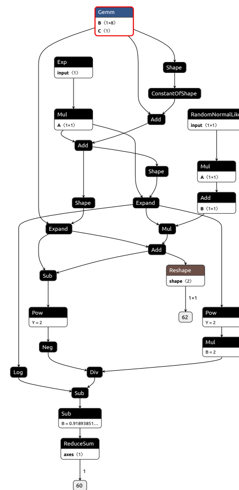
There are several NN implementations. BUT:

- mostly throughput optimized;
- targeting different platforms;
- real-time not in mind.

Most NN deployment to FPGA quantize $\rightarrow$ effect on RL algorithms?

Would be cool to directly translate agents into real-time, but we are not there yet.

An ONNX middle-layer for low-latency could be the solution!



Optimizations on x86: what the compiler does for you

Karlsruhe Institute of Technology

```
#define mat_N 64
std::vector<float> coeffs(mat_N*mat_N);
std::vector<float> data(mat_N);
std::vector<float> result(mat_N);
...

for (int i = 0; i < mat_N; ++i) {
    for (int j = 0; j < mat_N; ++j) {
        result[j] += data[i] *
        ↪  coeffs[j+i*mat_N];
    }
}

...
```

We need fast! What is the CPU doing?

Optimizations on x86: what the compiler does for you

Karlsruhe Institute of Technology

```
#define mat_N 64
std::vector<float> coeffs(mat_N*mat_N);
std::vector<float> data(mat_N);
std::vector<float> result(mat_N);
...

for (int i = 0; i < mat_N; ++i) {
    for (int j = 0; j < mat_N; ++j) {
        result[j] += data[i] *
        ↪  coeffs[j+i*mat_N];
    }
}

...
```

Not optimized: exec time 280 μs

```
# result[j] += data[i] * coeffs[j+i*mat_N];
...
movss xmm0, DWORD PTR [rax]
mulss xmm0, DWORD PTR -1780[rbp]
movss DWORD PTR -1780[rbp], xmm0
mov        eax, DWORD PTR -1764[rbp]
movsx rdx, eax
lea        rax, -1616[rbp]
mov        rsi, rdx
mov        rdi, rax
call  _ZNSt6vectorIfSaIfEEixEm
movss xmm0, DWORD PTR [rax]
addss xmm0, DWORD PTR -1780[rbp]
movss DWORD PTR [rax], xmm0
...
```

Optimizations on x86: what the compiler does for you

Karlsruhe Institute of Technology

```
#define mat_N 64
std::vector<float> coeffs(mat_N*mat_N);
std::vector<float> data(mat_N);
std::vector<float> result(mat_N);
...

for (int i = 0; i < mat_N; ++i) {
    for (int j = 0; j < mat_N; ++j) {
        result[j] += data[i] *
        ↪  coeffs[j+i*mat_N];
    }
}

...
```

Not optimized: exec time 280 μs

```
# result[j] += data[i] * coeffs[j+i*mat_N];
...
movss xmm0, DWORD PTR [rax]
mulss xmm0, DWORD PTR -1780[rbp]
movss DWORD PTR -1780[rbp], xmm0
mov       eax, DWORD PTR -1764[rbp]
movsx rdx, eax
lea       rax, -1616[rbp]
mov       rsi, rdx
mov       rdi, rax
call  _ZNSt6vectorIfSaIfEEixEm
movss xmm0, DWORD PTR [rax]
addss xmm0, DWORD PTR -1780[rbp]
movss DWORD PTR [rax], xmm0
...
```

Optimizations on x86: what the compiler does for you


```
#define mat_N 64
std::vector<float> coeffs(mat_N*mat_N);
std::vector<float> data(mat_N);
std::vector<float> result(mat_N);
...

for (int i = 0; i < mat_N; ++i) {
    for (int j = 0; j < mat_N; ++j) {
        result[j] += data[i] *
        ↪  coeffs[j+i*mat_N];
    }
}

...
```

Not optimized: exec time 280 μs

```
# result[j] += data[i] * coeffs[j+i*mat_N];
...
movss xmm0, DWORD PTR [rax]
mulss xmm0, DWORD PTR -1780[rbp]
movss DWORD PTR -1780[rbp], xmm0
mov       eax, DWORD PTR -1764[rbp]
movsx rdx, eax
lea       rax, -1616[rbp]
mov       rsi, rdx
mov       rdi, rax
call  _ZNSt6vectorIfSaIfEEixEm
movss xmm0, DWORD PTR [rax]
addss xmm0, DWORD PTR -1780[rbp]
movss DWORD PTR [rax], xmm0
...
```

Optimizations on x86: what the compiler does for you

Karlsruhe Institute of Technology

Not optimized: exec time 280 μs

```
# result[j] += data[i] * coeffs[j+i*mat_N];
...
movss xmm0, DWORD PTR [rax]
mulss xmm0, DWORD PTR -1780[rbp]
movss DWORD PTR -1780[rbp], xmm0
mov       eax, DWORD PTR -1764[rbp]
movsx rdx, eax
lea       rax, -1616[rbp]
mov       rsi, rdx
mov       rdi, rax
call   _ZNSt6vectorIfSaIfEEixEm
movss xmm0, DWORD PTR [rax]
addss xmm0, DWORD PTR -1780[rbp]
movss DWORD PTR [rax], xmm0
...
```

Optimized (`-O3 -march=native`): exec time 8.8 μs

```
# result[j] += data[j] * coeffs[j+i*mat_N];
.L19:
vmovaps ymm1, ymm0
vbroadcastss ymm0, DWORD PTR [rdx]
add          rax, 256
vfmadd231ps        ymm8, ymm0, YMMWORD PTR -256[rax]
... x5
vfmadd231ps        ymm2, ymm0, YMMWORD PTR -64[rax]
add          rdx, 4
vfmadd132ps        ymm0, ymm1, YMMWORD PTR -32[rax]
cmp          rcx, rax
jne          .L19
```

Optimizations on x86: what the compiler does for you

Karlsruhe Institute of Technology

Not optimized: exec time 280 μs

```
# result[j] += data[i] * coeffs[j+i*mat_N];
...
movss xmm0, DWORD PTR [rax]
mulss xmm0, DWORD PTR -1780[rbp]
movss DWORD PTR -1780[rbp], xmm0
mov       eax, DWORD PTR -1764[rbp]
movsx rdx, eax
lea       rax, -1616[rbp]
mov       rsi, rdx
mov       rdi, rax
call   _ZNSt6vectorIfSaIfEEixEm
movss xmm0, DWORD PTR [rax]
addss xmm0, DWORD PTR -1780[rbp]
movss DWORD PTR [rax], xmm0
...
```

Optimized (`-O3 -march=native`): exec time 8.8 μs

```
# result[j] += data[j] * coeffs[j+i*mat_N];
.L19:
vmovaps ymm1, ymm0
vbroadcastss ymm0, DWORD PTR [rdx]
add        rax, 256
vfmadd231ps        ymm8, ymm0, YMMWORD PTR -256[rax]
... x5
vfmadd231ps        ymm2, ymm0, YMMWORD PTR -64[rax]
add        rdx, 4
vfmadd132ps        ymm0, ymm1, YMMWORD PTR -32[rax]
cmp        rcx, rax
jne        .L19
```

Optimizations on x86: what the compiler does for you

Karlsruhe Institute of Technology

Not optimized: exec time 280 μs

```
# result[j] += data[i] * coeffs[j+i*mat_N];
...
movss xmm0, DWORD PTR [rax]
mulss xmm0, DWORD PTR -1780[rbp]
movss DWORD PTR -1780[rbp], xmm0
mov      eax, DWORD PTR -1764[rbp]
movsx rdx, eax
lea      rax, -1616[rbp]
mov      rsi, rdx
mov      rdi, rax
call  _ZNSt6vectorIfSaIfEEixEm
movss xmm0, DWORD PTR [rax]
addss xmm0, DWORD PTR -1780[rbp]
movss DWORD PTR [rax], xmm0
...
```

$\approx$ 30 improvement by doing 8 mult per cycle!

Optimized (`-O3 -march=native`): exec time 8.8 μs

```
# result[j] += data[j] * coeffs[j+i*mat_N];
.L19:
vmovaps ymm1, ymm0
vbroadcastss ymm0, DWORD PTR [rdx]
add        rax, 256
vfmadd231ps        ymm8, ymm0, YMMWORD PTR -256[rax]
... x5
vfmadd231ps        ymm2, ymm0, YMMWORD PTR -64[rax]
add        rdx, 4
vfmadd132ps        ymm0, ymm1, YMMWORD PTR -32[rax]
cmp        rcx, rax
jne        .L19
```

Optimizations on x86: what the compiler does for you

Karlsruhe Institute of Technology

Not optimized: exec time 280 μs

```
# result[j] += data[i] * coeffs[j+i*mat_N];
...
movss xmm0, DWORD PTR [rax]
mulss xmm0, DWORD PTR -1780[rbp]
movss DWORD PTR -1780[rbp], xmm0
mov       eax, DWORD PTR -1764[rbp]
movsx rdx, eax
lea       rax, -1616[rbp]
mov       rsi, rdx
mov       rdi, rax
call  _ZNSt6vectorIfSaIfEEixEm
movss xmm0, DWORD PTR [rax]
addss xmm0, DWORD PTR -1780[rbp]
movss DWORD PTR [rax], xmm0
...
```

$\approx$ 30 improvement by doing 8 mult per cycle!

Optimized (`-O3 -march=native`): exec time 8.8 μs

```
# result[j] += data[j] * coeffs[j+i*mat_N];
.L19:
vmovaps ymm1, ymm0
vbroadcastss ymm0, DWORD PTR [rdx]
add          rax, 256
vfmadd231ps         ymm8, ymm0, YMMWORD PTR -256[rax]
... x5
vfmadd231ps         ymm2, ymm0, YMMWORD PTR -64[rax]
add          rdx, 4
vfmadd132ps         ymm0, ymm1, YMMWORD PTR -32[rax]
cmp          rcx, rax
jne          .L19
```

Compiler made assumptions on HW!

AI Engine compiler

Karlsruhe Institute of Technology

Do these free optimizations work also for the Versal AI Engines?

AI Engine compiler


```
for (int i = 0; i < 64; i++)
    for (int j = 0; j < 64; j++)
        res[j] += data[i] * coeffs[j+i*64];
```

```
NOP;          LDB r5, [sp, #-12]; MUL r2, r6, r7
NOP;          ASHL r1, r13, r9
NOP;          MUL r5, r12, r5
ZE.16 r0, r2
NOP;          MUL r13, r5, r13
```

AI Engine compiler

Karlsruhe Institute of Technology

```
v8float* restrict data_v8   = (v8float*) data;
v8float* restrict coeffs_v8 = (v8float*) coeffs;
v8float* restrict res_v8    = (v8float*) res;

for (int i8 = 0; i8 < 8; i8++)
for (int el = 0; el < 8; el++)
for (int j8 = 0; j8 < 8; j8++)
    res_v8[j8] = fpmac(
        res_v8[j8],
        data_v8[i8],
        el,
        0x0,
        coeffs_v8[j8+el*8+i8*64],
        0x0,
        0x76543210
    );
```

```
1  VLDA wd1, [sp, #-64]; NOP;  VMOV wd1, wr1;
   ↪  VFPMAC wd1, r5, wd1, ya, r11, cl2, wc0, #0, cl0,
   ↪  #0, cl1
2  VLDA wc0, [p5], m0;   NOP;  VMOV wr1, wr2
3  NOP;                  NOP;  VMOV wr2, wd1;
   ↪  VFPMAC wr3, r0, wr3, ya, r11, cl2, wc1, #0, cl0,
   ↪  #0, cl1
4  NOP;                  NOP;  NOP;
   ↪  VFPMAC wr2, r2, wr2, ya, r11, cl2, wc1, #0, cl0,
   ↪  #0, cl1
5  NOP;                  NOP;  VLDA.SPIL wd1, [sp,
   ↪  #-160];   VFPMAC wd1, r1, wd1, ya, r11, cl2, wc0,
   ↪  #0, cl0, #0, cl1
```

AI Engine compiler

Karlsruhe Institute of Technology

```
v8float* restrict data_v8   = (v8float*) data;
v8float* restrict coeffs_v8 = (v8float*) coeffs;
v8float* restrict res_v8    = (v8float*) res;

for (int i8 = 0; i8 < 8; i8++)
for (int el = 0; el < 8; el++)
for (int j8 = 0; j8 < 8; j8++)
    res_v8[j8] = fpmac(
        res_v8[j8],
        data_v8[i8],
        el,
        0x0,
        coeffs_v8[j8+el*8+i8*64],
        0x0,
        0x76543210
    );
```

Conclusion

Specialized optimization required! The compiler does not save you!

The KINGFISHER RL platform

Experience accumulator

Real-Time inference BUT Offline/Batched training

The KINGFISHER RL platform

Karlsruhe Institute of Technology

Experience accumulator

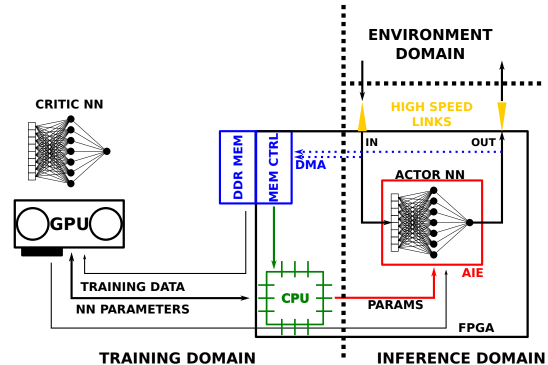
Real-Time inference BUT Offline/Batched training

Pros:

- + "easy" real-time;
- + can use complex training algorithms;
- + can use GPUs and other accelerators;
- + training time reward definition™.

Cons:

- - data inefficient;
- - actor design is critical;
- - training overhead.



Less is more

Karlsruhe Institute of Technology

An example: control of the microbunching instability at KARA
Environment $\rightarrow x_i$ Coherent Sychrotron Radiation power each turn

Initial approach

$$O = \{\mu_{\text{CSR}}, \sigma_{\text{CSR}}, m_{\text{trend}}, A_{\text{FFT max}}, f_{\text{FFT max}}, \Delta_\theta\}$$

$$A = \{A_{\text{mod}}, f_{\text{mod}}\}$$

Issue! FFT and cross-correlation needed with $O(10\ \mu s)$ latency

Less is more

Karlsruhe Institute of Technology

An example: control of the microbunching instability at KARA
Environment $\rightarrow x_i$ Coherent Sychrotron Radiation power each turn

Initial approach

$$O = \{\mu_{\text{CSR}}, \sigma_{\text{CSR}}, m_{\text{trend}}, A_{\text{FFT max}}, f_{\text{FFT max}}, \Delta_\theta\}$$

$$A = \{A_{\text{mod}}, f_{\text{mod}}\}$$

Issue! FFT and cross-correlation needed with $O(10\ \mu s)$ latency

New approach

$$O = \{N \text{ latest } x_i\}$$

$$A = \text{action or delta-action}$$

Hardware friendly! Still rich of information

Less is more

Karlsruhe Institute of Technology

An example: control of the microbunching instability at KARA
Environment $\rightarrow x_i$ Coherent Sychrotron Radiation power each turn

Initial approach

$$O = \{\mu_{\text{CSR}}, \sigma_{\text{CSR}}, m_{\text{trend}}, A_{\text{FFT max}}, f_{\text{FFT max}}, \Delta_\theta\}$$

$$A = \{A_{\text{mod}}, f_{\text{mod}}\}$$

Issue! FFT and cross-correlation needed with $O(10\ \mu s)$ latency

New approach

$$O = \{N \text{ latest } x_i\}$$

$$A = \text{action or delta-action}$$

Hardware friendly! Still rich of information

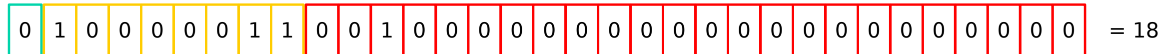
Simplify problem definition with real-time Digital Signal Processing

Choose smaller models ($\approx$ 128 fully connected neurons)

Example: the incredible efficiency of ReLU

ReLU is easy and efficient to implement both in floating-point and integer representations

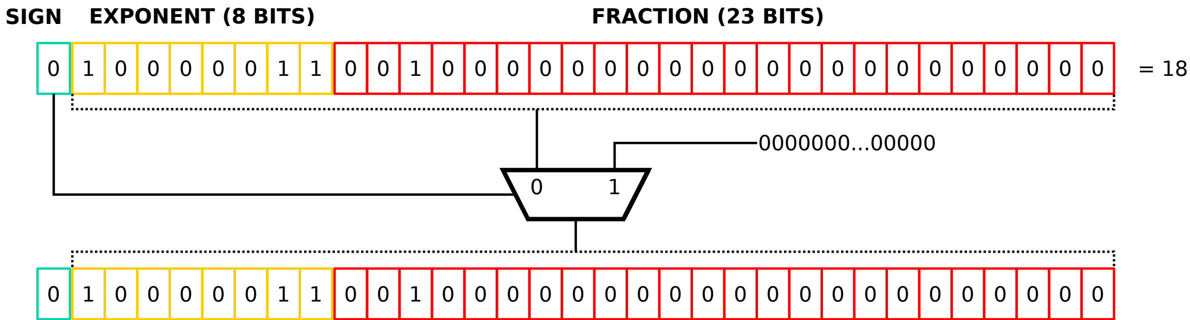
SIGN **EXPONENT (8 BITS)** **FRACTION (23 BITS)**



$$(-1)^0 \times 2^4 \times 1.001_2$$

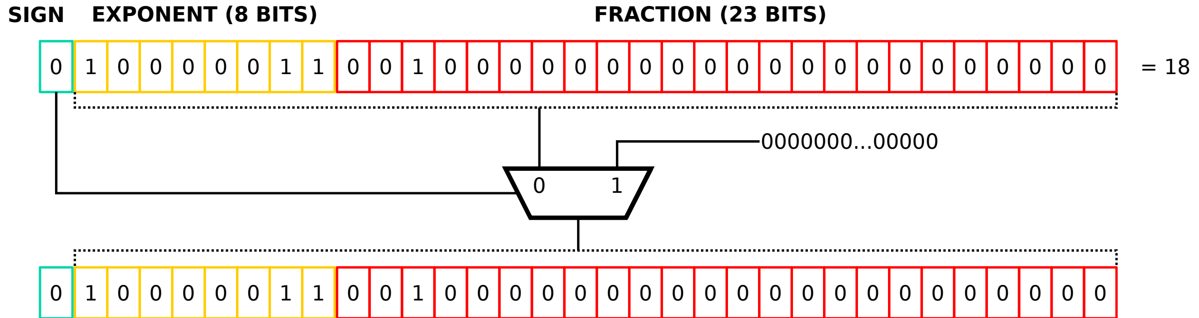
Example: the incredible efficiency of ReLU

ReLU is easy and efficient to implement both in floating-point and integer representations



Example: the incredible efficiency of ReLU

ReLU is easy and efficient to implement both in floating-point and integer representations



Extremely fast O(ns) and parallellizable operation

Example: the incredible efficiency of ReLU

Karlsruhe Institute of Technology

ReLU is easy and efficient to implement both in floating-point and integer representations

SIGN EXPONENT (8 BITS) FRACTION (23 BITS)

0 1 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 = 18

0000000...00000

0 1

0 1 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Extremely fast O(ns) and parallellizable operation

On AIE ReLU$(x) = max\ \{x, 0\}$ single instruction on 8 values

Is Real-Time RL what you need?

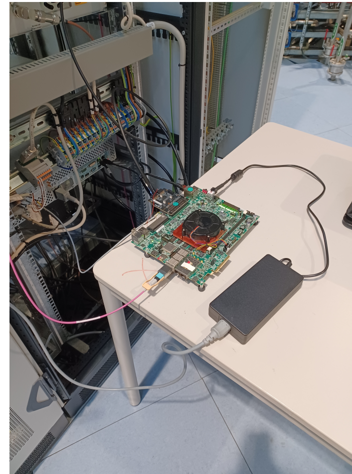
Karlsruhe Institute of Technology

Requirements:

- low-latency diagnostics & actions;
- implementable policy (no μs ChatGPT, sorry)
- high data production rate.

Pros & Cons:

- \+ no sim2real issues;
- \+ can directly try on environment;
- \+ for complex dynamics $\rightarrow$ faster than simulation;
- \- choice of policy is limited;
- \- careful observation and action design;
- \- fast safety measures;
- \- not everything can be done "fast".



READY TO DEPLOY!

Conclusion

Karlsruhe Institute of Technology

- μs Real-Time RL is a viable option
- Its performance is problem dependent
- FPGAs and Heterogeneous platforms are the key
- Hardware aware problem design is fundamental

Alex Blechman
@AlexBlechman

Programming is chaotic magic. There are no rules. You ask a game dev "Can the player summon a giant demon that bursts from the ground in an explosion of lava?" and they'll say "sure, that's easy" and then you'll ask "can the player wear a scarf?" and they'll go "oof"

Sounds interesting? Let's find more applications!